

Artificial Neurogenesis in the Context Continual Learning

MASTER MATHEMATICS, VISION ET APPRENTISSAGE

MATHIS REYMOND

Supervisor:	Prof. Melika Payvand
Institution:	Federal Institute of Technology Zurich
Laboratory:	Institute of Neuroinformatics

April 1st 2024 - August 15th 2024

Abstract

In this thesis, we introduce and develop a novel approach to artificial neurogenesis in the context of continual learning. It involves the generation and integration of new neural units within an artificial neural network that is being trained, enabling it to adapt and learn without forgetting previous knowledge. After stating the specific hypotheses of our framework and comparing our method to the literature, we introduce a formalism to bring to light the various ways hyperparameters optimizations can be conducted in continual learning. Additionally, we motivate and introduce a validation paradigm specific to continual learning that we leverage to demonstrate overfitting on benchmarks. An important part of this work focuses on a critical analysis of a whole branch of the literature, including our own approach. After putting forward the assumptions implicitly made by numerous approaches available in the literature, this critic aims to show how they lead to inherent limitations embedded in their core.

The code is available partly on [my GitHub](#) and partly on the team's [GitHub](#).

Contents

1	Introduction	2
1.1	Continual learning	2
1.2	Approaches to continual learning	2
1.3	Framework specification	3
2	GroHess	3
2.1	Motivation	3
2.2	Description	4
2.3	Baselines	4
3	Main experiment	6
3.1	Permuted MNIST	6
3.2	Training protocol	6
4	Results	7
5	Paradigmatic discussion	9
5.1	Hyperparameters optimizations	9
5.2	Introduction of validation benchmarks	11
6	Critical comparison to literature	12
6.1	Trade-offs in continual learning	12
6.2	Geometrical argument	13
6.3	Combinatorial argument	15
7	Conclusion	16
8	Acknowledgment	17

1 Introduction

1.1 Continual learning

We, humans are able to learn a wide variety of tasks throughout our lives. We can learn to play the piano, to speak a new language, to ride a bike, to cook a new recipe, etc. We can even learn to perform several tasks at the same time, such as playing the piano and singing simultaneously. This ability to learn a sequence of tasks without forgetting the previously learned tasks is called continual learning. Continual learning is a fundamental aspect of human intelligence, and it is a key challenge in artificial intelligence. Indeed, when an artificial model using error back-propagation [24] is assigned to learn a sequence of tasks, its final state after training on a task is perceived as an initial state to train on the next task. Without paying attention to this fact, the state will simply be overwritten during the training procedure on the new task, leading to catastrophic forgetting of the previous task. This is a major limitation of back-propagation on neural networks, and it is a key challenge in artificial intelligence. In this work, we will introduce a novel approach to continual learning that is based on the idea of dynamically growing neurons in a neural network and we will discuss the problems associated to continual learning.

In a very broad sense, one can define continual learning as

*the problem of learning several tasks
sequentially.*

Although this assumption is sometimes relaxed in the literature [23], the fact that tasks have to be learnt *sequentially* implies that once the training procedure on a task is done, its associated training data cannot be used again.

Continual learning comes in different flavors that vary depending on the working assumptions one makes, such as whether or not the model knows when it starts training on a new task or whether or not all tasks involve the same classes, in the context of classification. Some of the most common frameworks include the followings [9] [31].

In *Task Incremental Learning*, the model is trained sequentially on a series of tasks, with each task involving a different set of data or objectives, such as sentiment analysis and then object classification within images. The model knows which task it is dealing with both during training and inference, and it often has task-specific components or outputs. The model loses ac-

cess to the old data once it starts training on a new task. This framework is very general and can be applied to a wide range of problems. In *Domain Incremental Learning* framework, the model faces a sequence of tasks that involve the same objective but in different domains. An example of domain incremental use case would be the classification of images of dogs and cats. Each task would require the same thing, classifying images of dogs from images of cats, but for each task, there would only be images of cats of the same breed and dogs of the same breed. A first task might involve classifying bulldogs and Chartreux cats. And a second task might involve classifying huskies and American Curl cats. During inference, the model is not informed which domain it is operating in and must generalize across these domains. Finally, *Class Incremental Learning* involves learning to solve classification tasks sequentially, where new classes are added in each task. This framework excludes all form of task identifiers. The model is expected to classify data from any of the classes it has learned so far, even as new classes are introduced. This is often considered a very challenging form of continual learning, as the model must avoid catastrophic forgetting while continuously expanding its ability to recognize new classes.

Literature also acknowledges other frameworks such as *Lifelong Learning* [20][13][34], and other classifications of these frameworks [review1][12].

1.2 Approaches to continual learning

The problem of continual learning has been heavily investigated in the past decade, leading to the development of various approaches to mitigate catastrophic forgetting. These approaches can be broadly categorized into regularization-based, replay-based and architectural.

Regularization-based approaches, which constrain the learning process through a regularization term embedded in the loss function, aim at preventing drastic changes to model parameters that are critical for previous tasks. Foundational methods to this category of approaches include Elastic Weight Consolidation (EWC) [11][1], which estimates the importance of each parameter and penalizes changes accordingly, and Synaptic Intelligence (SI) [35], which dynamically accumulates information about parameters importance throughout training of all the tasks.

Another important group of approaches is *replay-based learning* [29][28][23][22][30][17][27], which involves storing or generating data from previous tasks to revisit

during the training of new tasks. This helps maintaining performance on older tasks by refreshing the model’s memory of past data. Methods like Experience Replay [23] store a small buffer of past samples which are forwarded to the model when training on new tasks, while Generative Replay [28][22][30][27] uses an external generative model to recreate as many data from previous tasks as needed.

Finally, *architectural approaches* modify the model’s structure to accommodate new tasks while preserving knowledge from previous ones. Methods like Progressive Neural Networks (PNN) [25] add new sub-networks for each task, allowing to forward knowledge transfer from old tasks dedicated sub-networks, while keeping the new ones relatively isolated. Conversely, Mixture of Expert Models [8][5] leverage a gating system which is designed to identify and distribute tasks to expert sub-networks.

However, these categories do not constitute a proper partition of the literature as they are not mutually exclusive. Approaches such as Architectural and Regularization 1 (AR1) [18] falls both in regularization-based and architectural categories. Additionally, these categories do not encompass certain methods such as adversarial approaches [4].

1.3 Framework specification

Given the variety of existing frameworks and approaches, it is essential to refine and specify our own. Here we present our specific framework and approach within the context of the previously discussed literature. While the earlier subsections categorized various approaches to continual learning to provide a concise overview of the literature, we will focus on the specific assumptions that underpin our methodology, as these are crucial for laying properly the foundation of our analysis.

First, we will focus on solving classification tasks involving the same set of classes. Each task differs from one another by a specific domain of the input data. We assume that the model is aware when transitioning to the training of a new task. However, during inference time, the model does not know which task each sample belongs to. Each task must be handled strictly sequentially, meaning we do not permit any use of data from previous tasks. This is conceptually closest to domain incremental learning. Additionally, we assume the total number of tasks is unknown and cannot be leveraged during the learning process.

Second, as for the methods, our approach will be *modular*, meaning it will break down the learning pro-

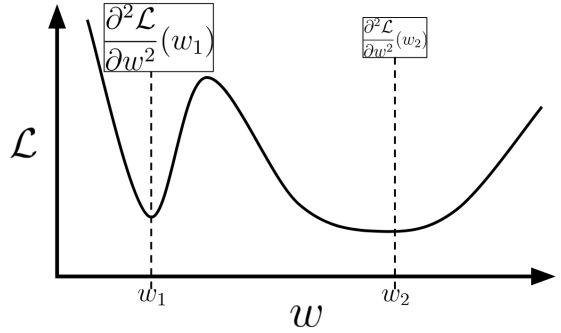


Figure 1: Illustration of our sense of importance of weights. When the second derivative of the loss with respect to a weight is high, the weight is considered important as modifying it even a little bit would lead to a significant change in the loss. However, when the second derivative is low, modify the weight has little impact on the loss.

cess into sub-problems and explicitly leverage dedicated modules to address these sub-problems. In other words, we will identify and manage certain components of our neural network that encapsulate high-level features, allowing us to handle them in a targeted manner. These components, or modules, may consist of independent sub-networks, sub-networks that share some parameters, or groups of neurons distributed within the entire network.

On the top of the modularity of our approach, we make two additional assumptions regarding the way we will process information. First, we will work with *unconstrained* architectures, meaning that we will not impose any architectural constraint on the model that could allow the processing or implicit remembering of the specificities of the past tasks. To this regard, convolutional or graph neural networks are out of our scope. We will also work with *memoryless* strategies, meaning that in addition to banning the use of data from previous tasks, we will not allow to leverage a replay buffer or generative rehearsal module.

2 GroHess

2.1 Motivation

To address the problem of Continual Learning, one approach is to train independent specialized modules and a gating system that learns to recognize tasks, similar to Mixture of Experts Models [8][5]. However, the tasks to be learned often share common features that we would like to leverage, so that only the new features of a task are learned without forgetting the old

ones. The emergence of such features within a neural network relies on constraints, starting with imposing small size of the model. Therefore, we would rather start training with a model of moderate size and expand it as more tasks are introduced.

Such an approach presents several challenges. First, it requires to determine which important features to retain. Then, we must figure out how to retain them. Finally, we need to determine how to use them to handle new tasks. To achieve this, we introduce GroHess, a new approach to architecture growth in the context of continual learning. GroHess uses the Hessian to identify which weights are crucial for previous tasks and which can be adjusted for new tasks. Additionally, when modification of a weight that was important for a previous task is necessary, GroHess instead adds a new weight to preserve the integrity of the original one. The importance of a weight for a given is determined by the second derivative of the loss on this task with respect to the weight, once training on this task is completed. This is illustrated in Fig. 1. When the second derivative of the loss with respect to a weight is high - such as in w_1 - the weight is considered important as modifying it even a little bit would lead to a significant change in the loss. Conversely, when the second derivative is low - such as in w_2 - modifying the weight has little impact on the loss. This allows us to determine which weights are important and which are not. Then, when back-propagation encourages the modification of an important weight, we refrain from updating this weight, but add a new weight by growing a new neuron and performing the update on that weight. Overall, the growing process involves selecting key neurons that are both relevant for a previous task and sensitive during learning of the current task, advocating for the need to grow a new neuron.

2.2 Description

More specifically, we control the growth of the model through two percentiles : a gradient percentile that regulates which weights are considered to have a high gradient, relatively to the other weights, and a Hessian percentile that accounts for weights for which the derivative of the loss with respect to them is high.

Once the training procedure on a task is done, we compute the second derivative of the loss with respect to each weight through a random portion of the data batches of the task (10%). When the second derivative of a given weight with respect to the loss falls above the hessian percentile, it means that modifying this

weight will strongly disturb what has been learnt on the task. Thus, we compute a binary mask recording which weights are important for the task according to the second derivative. Conversely, when training on a latter task, if the loss starts to stagnate, we take a look at the last computed gradient with back-propagation. If the gradient of a given weight falls above the gradient percentile, it means that according to back-propagation, this weight has to be strongly updated. As a consequence, when both the first and the second derivatives of a given weight are high, we have a tension that leads us into growing a new neuron, to accommodate both back-propagation on the task at hand and the preservation of the knowledge of the previous task. The weight we grow from is then frozen; it won't be updated ever again, but we will keep back-propagating gradients through it.

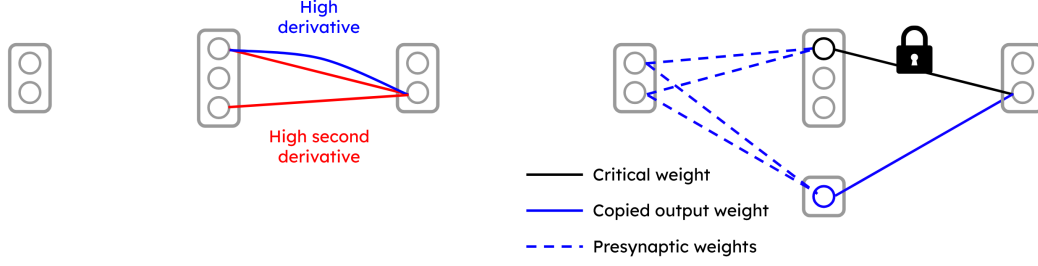
Every time a neuron is grown, it has to be initialized, which means that 6 choices have to be made: the value of its bias, the value of the incoming weights, the value of the outgoing weights and their respective gradients. The impact of a growth is to allow both the preservation of the knowledge of the learnt tasks and the learning of the new task. As a consequence, we mostly copy the neuron we are growing from, and we only set to zero the outgoing weights whose first and second derivatives were not both high. This choice is motivated by the fact that we do not want the grown neuron to disrupt the learning process. See Alg. 1

2.3 Baselines

To evaluate the effectiveness of our method, which incorporates both a Hessian-inspired mechanism for determining weight importance and a growth mechanism for adding new neurons to the model, we introduce four baselines. All the baselines share the same optimizable hyperparameters and the values of these hyperparameters are optimized independently for each baseline.

In a first "vanilla" baseline, we define a model in the same initial configuration as the one used to optimize with GroHess. During the training procedure through the tasks, no additional neurons are grown, and no special mechanism is used to adjust or freeze weights according to their importance for the tasks. The model remains at its original size throughout the training process.

We also train a second "vanilla" baseline, but the initial size of its hidden layers is that of the average final hidden layer sizes when training with GroHess. So, from the beginning, this baseline has the average ar-



(a) The Hessian mask is computed at the end of training on task $n - 1$. In this illustration, an important weight is represented in blue. And the Gradient mask is computed during training on task n . In this illustration, the two weights in red have to be strongly updated according to back-propagation. Here, there is an important weight that has to be strongly updated, so it is a critical one. As a consequence, GroHess will grow a new neuron.

(b) The critical weight is frozen by GroHess and the back-propagation update that should have been performed on the critical weight is applied to a copy of it, made through the addition of a new neuron. The critical weight will remain frozen now on, while the grown neuron will be optimized as any other neuron during the rest of the training process on task n .

Figure 2: Illustration of the GroHess procedure.

Algorithm 1 Training GroHess on n tasks

```

1: Input:
2: - Model  $\Sigma_{ini}$ 
3: - List of  $n$  tasks  $L = [(X_1, Y_1), \dots, (X_n, Y_n)]$ 
4: - Hessian percentile  $\tau_{hessian}$ 
5: - Gradient percentile  $\tau_{gradient}$ 
6: Output: Trained model  $\Sigma_{fin}$  solving each task
7:
8: Step 1: Train  $\Sigma_{ini}$  on  $(X_1, Y_1)$ 
9: Step 2: Compute  $\nabla_{\Sigma_{ini}}^2 \mathcal{L}_{(X_1, Y_1)}$  for each weight and define the binary mask  $\mathcal{M}_{hessian}$  using  $\tau_{hessian}$ 
10: for each remaining task  $(X_i, Y_i)$  do
11:   for each batch  $(x_i, y_i)$  in  $(X_i, Y_i)$  do
12:     if loss stagnates then
13:       Step 3: Compute  $\nabla_{\Sigma_{ini}} \mathcal{L}_{(x_i, y_i)}$  and define the binary mask  $\mathcal{M}_{gradient}$  using  $\tau_{gradient}$ 
14:       Step 4: Compute  $\mathcal{M}_{overlap}$  as the element-wise product of  $\mathcal{M}_{hessian}$  and  $\mathcal{M}_{gradient}$ 
15:       Step 5: Grow as many neurons on  $\Sigma_{ini}$  as  $\mathcal{M}_{overlap}$  has columns with at least a value equal to 1
16:       Step 6: Freeze the non-zero weights in  $\mathcal{M}_{overlap}$ 
17:     end if
18:     Step 7: Perform training step on  $(x_i, y_i)$ 
19:   end for
20:   Step 8: Compute  $\nabla_{\Sigma_{ini}}^2 \mathcal{L}_{(x_i, y_i)}$  and update  $\mathcal{M}_{hessian}$  using  $\tau_{hessian}$ 
21: end for

```

chitecture of a model trained with GroHess.

Additionally, we design a third baseline, where we perform growth without GroHess, during the training procedure. This baseline involves starting the training process on the first task with a model of the same size as the one used with GroHess. However, every time the model moves to training on a new task, it blindly adds new neurons to each layer, randomly initialized, and without freezing any weight. This baseline helps evaluate the benefits of our approach of defining weights importance through Hessian coefficients.

Finally, our fourth and last baseline uses GroHess to freeze important weights without triggering the growing mechanism. This baseline is initialized in the same way as the second one, however, we leverage GroHess to freeze important weights during the training process, but without growing new neurons as GroHess would normally do. This baseline is supposed to isolate the effectiveness of the growing mechanism to improve performances.

These baselines allow us to thoroughly analyze the contribution of the two main mechanisms of our approach, namely the Hessian-inspired weight importance mechanism and the growth mechanism, independently and in combination.

3 Main experiment

3.1 Permuted MNIST

A permuted MNIST (p-MNIST) dataset [7] is a variant of the classic MNIST dataset [15] where images are transformed through a fixed random permutation of pixel positions, creating a shuffled version of each original image. The label associated with each image remains the same, meaning the digits themselves are unchanged, but their visual structure is modified. Then, a p-MNIST task is defined as solving the classification problem associated to a p-MNIST dataset. As each p-MNIST dataset is generated by a different permutation of MNIST dataset, there are $784!$ possible p-MNIST datasets and as many p-MNIST tasks. Finally, a p-MNIST benchmark is created by selecting a subset of these tasks. See Fig. 3 for illustration.

Permuted MNIST is particularly significant in the context of continual learning as it offers the opportunity to generate arbitrary long sequences of tasks with a fixed number of classes and a fixed number of samples per class. Additionally, users have a great control over the difficulty of a benchmark, which is directly as-

sociated with the similarity of the permutations used to generate the tasks of the benchmark. Finally, the p-MNIST benchmark is a well-established benchmark in the continual learning literature, which allows for comparison of different approaches.

However, literature has come with a lot of criticism about the p-MNIST benchmarks [9][6]. First, they are rather simple benchmarks, which do not reflect the complexity of real-world tasks. Second, the random permutations break the spatial structure of images we can find in the real world, which means that the p-MNIST benchmarks are not well-suited to evaluate methods that require spatial reasoning on tasks such as object detection or segmentation. In particular, the permutations of pixels disrupt the assumption of locality and translation invariance which are at the basis of convolutional neural networks. As a consequence, approaches leveraging convolutional neural networks are disadvantaged on p-MNIST benchmarks.

Despite this criticism, we have chosen to use p-MNIST benchmarks. They are relevant for our study of GroHess and the critic we develop in the latter sections as their very simplicity allows us to shed light on the fundamental principles of our exploratory method and continual learning. Additionally, the ability to generate not only several tasks, but also several comparable benchmarks will allow us to motivate and introduce a validation paradigm in continual learning.

3.2 Training protocol

Our model consists of a multi-layer perceptron with 2 hidden layers (or 3 layers in total). Input and output sizes are 784 and 10 respectively. Each hidden layer initially contains 300 neurons. The growth of the model happens from the output layer, which means that when a weight has both a high first and second derivatives, we grow a neuron on the layer preceding it (in the forward sense). We control the growth of the model through two percentiles : the gradient percentile $\tau_{gradient}$ and the Hessian percentile $\tau_{hessian}$. When both first and second derivative of the loss with respect to a weight are above the gradient and Hessian percentiles, respectively, we grow a new neuron.

We train our models following GroHess algorithm on a sequence of 10 p-MNIST tasks, each with the usual 10 MNIST classes. For each task 80% of the train set is used for training, 20% is used for validation and 100% of the test set is used to test the model. We train GroHess with the Adam optimizer and the cross-entropy loss. For each task, the number of epochs and the

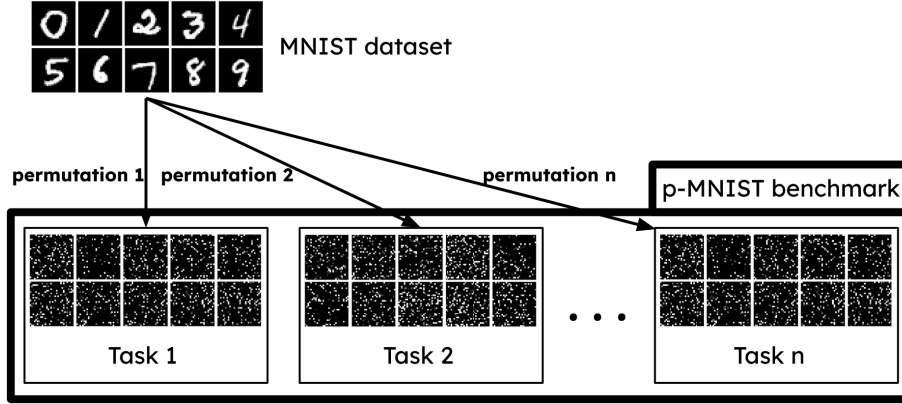


Figure 3: Illustration of the process of creating a p-MNIST benchmark. A p-MNIST dataset is a variation of the classic MNIST dataset, where pixel positions are shuffled by a fixed random permutation, altering the image’s visual structure while keeping the labels intact. A p-MNIST task involves solving the classification problem for such a dataset, and a p-MNIST benchmark is created by selecting a subset of these tasks.

learning rate are left as optimizable hyperparameters. We use a batch size of 128. See Tab. Table 1 for the complete list of hyperparameters.

Hyperparameters (HPs)	
Fixed HPs	Value
Number of layers	3
Initial hidden layers size	300
Batch size	128
Optimizer	Adam
Loss	Cross-entropy
Growth happens from	Output
Gradient percentile	0.98
Hessian percentile	0.98
Growth trigger	20
Optimized HPs	Range
Learning rate	1e-5 - 2e-3
Number of epochs	2 - 10

Table 1: Hyperparameters value or range

As the training loop is called once for each task, it means that a learning rate and number of epoch have to be found through hyperparameter optimization (HPO) for each task. HPO is performed using the Optuna library [2] with the Tree Parzen Estimator (TPE) algorithm. We use the validation set to evaluate the performance of the model and to optimize the hyperparameters. We use the average accuracy over all tasks as the metric to optimize. We perform HPO for each task sequentially, meaning that once the best hyperparameter values for a task are found, we retrain the model with them before moving to the next task. We repeat this process 10 times with random seeds 88 to 92 and we report the average test accuracy over all

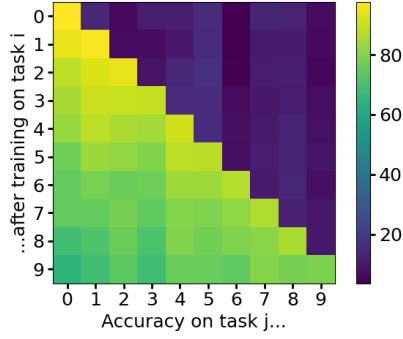
runs.

4 Results

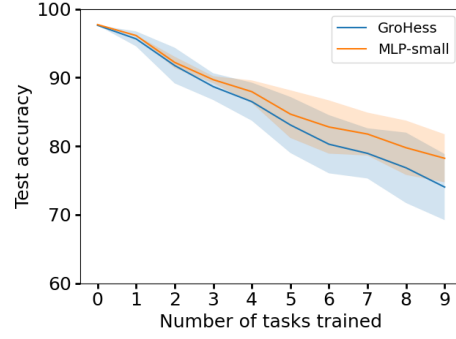
In Fig. 4a, we show the test accuracy on each task as GroHess moves forward through the tasks. This visualization provides a better insight than aggregated metrics as it allows us to inspect directly the performance of the model on each task during training. An interesting feature of our method is that the accuracies below the diagonal of the matrix, on the same line are relatively equal, which means that our GroHess is able to retain knowledge of the previous tasks without completely overwriting them. Additionally, we observe that column-wise, the accuracy is decreasing, which is reasonable: as the model learns tasks, it has to remember more past tasks, which makes the learning process harder.

Additionally, Fig. 4b shows the average test accuracy on tasks trained so far. In other words, the value associated to abscissa i is the average of the accuracies below the diagonal of the i^{th} line of the matrix in Fig. 4a. This plot is a more classical way to represent the performance of the model as it moves forward through the tasks, as it shows the ability of the approach to learn the first tasks of the benchmark. The last point of this plot represent the average test accuracy over all tasks, once the model has been trained on all of them.

Additionally, we present the results obtained with GroHess along with a comparison to literature in Fig. 5a. We also compare our approach to the baselines we introduced in the previous section. Each dot in this plot represent the average accuracy over all tasks of

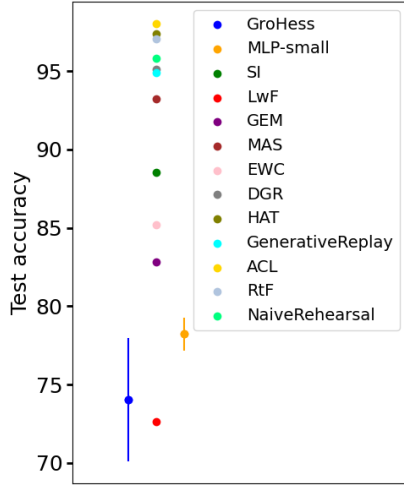


(a) Test accuracy on each task as GroHess move forward through the tasks.

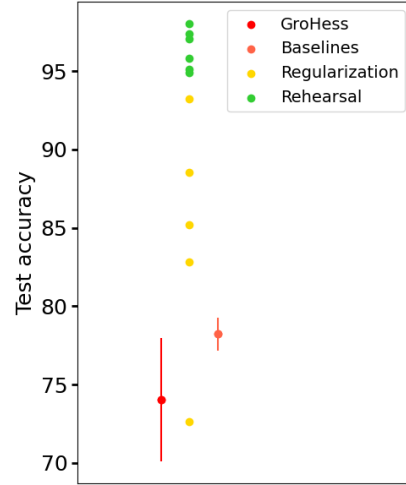


(b) Average test accuracy on tasks trained so far.

Figure 4: Evolution of the average test accuracy during training. The plots show the average test accuracy over all runs on seed 88 to 92. Fig. 4b also represents the standard deviation over runs.



(a) Our results, along with results collected in the literature on the 10 tasks p-MNIST benchmark [35][16][17][3][11][27][26][4][30].



(b) Same results as in Fig. 5a, but grouped by type of approach.

Figure 5: Results

a p-MNIST benchmark after training on all of them sequentially. For both our methods and the methods reproduced, the error bars represent the standard deviation over runs on distinct seeds. The permutations used to generate the p-MNIST tasks are the same for all the methods we ran, namely the first permutation as suggested by the random module of Numpy when provided 10 different seeds.

Then, we group the results by type of approach in Fig. 5b. We identify detached groups. The methods relying on replay-based learning or rehearsal are performing much better than the other methods. The performances of the regularization methods are more spread, but they are generally better than our baselines and GroHess. Interpretation of the results is available through section 6.

5 Paradigmatic discussion

5.1 Hyperparameters optimizations

In section 3.2, where the training protocol is introduced, we suggested to have only two hyperparameters, the number of epochs and the learning rate. However, as we mentioned, we performed one HPO for each task we train our model on. As a consequence, despite having two hyperparameters, we end up with 10 values for each of them, which means that the whole training procedure on the benchmark carries 20 hyperparameters.

Continual learning enables several ways to manipulate hyperparameters. In this subsection, we will discuss various approaches to manipulate them. This will help us understand that the various approaches to hyperparameter optimization (HPO) in continual learning are not equivalent: some are reasonable, others are problematic, and some violate fundamental assumptions of continual learning. Please note that the following discussion is independent of the specific optimization objective, focusing instead on the design of the HPO process itself.

Let us introduce a hyperparameter λ . One could set this hyperparameter to be the same for all the tasks, which would result in a total of 1 hyperparameter. To set the value of this hyperparameter, one could perform a single HPO on the first task, and keep using the same value for the forthcoming task. However, when proceeding that way, the hyperparameter is not optimized for each task, but only for the first one. Therefore, this way of performing HPO is not suited for our continual learning framework. Alternatively, in order

to have a single value of λ for the whole benchmark while optimizing it for all the tasks, one could perform a single HPO on the whole benchmark. It means that the HPO algorithm would suggest a value for λ that would be used to train all the tasks. The trial would end once the model has been trained on all the tasks. At the end of such an HPO, we would end up with a single value of λ that would be tuned for all the tasks. However, optimizing in such a way requires to have access to all the tasks at the same time, which breaks the continual learning framework. Indeed the value of λ used when training on the first task has to be decided before accessing data from the latter tasks. Overall, this way of performing HPO is highly problematic as it causes data leakage. That is why we refer to it as a cheated HPO with respect to continual learning framework. See Tab. Table 2 top-left.

Alternatively, one could set this hyperparameter to a specific value λ_i for each task i , this would require to perform an HPO on each task and would result in a total of n hyperparameters. This way of performing HPO is suited for continual learning framework, as tasks are treated strictly sequentially. This is how we proceeded when training with GroHess. See Tab. Table 2 bottom-center. As optimizing for the value of λ_{i+1} requires to have the value of λ_i , we refer to this approach as greedy HPO.

Remarkably, one could merge both of the previous approaches in a cheated HPO with n parameters. This would require to perform a single HPO on the whole benchmark, but instead of suggesting a single value of λ at each trial, the HPO algorithm would suggest values $\lambda_1, \dots, \lambda_n$, one for each task i . This way of performing HPO is even more problematic than the first way as it combines data leakage and a rather high number of hyperparameters. See Tab. Table 2 top-center.

Finally, we will see in this paragraph how some methods in the literature can enable for HPO method that has not 1, not n , but $O(n^2)$ values of λ . Indeed, some methods in the literature introduce a hyperparameter λ that is explicitly used to make trade-offs between how much the model should learn from the new task versus how much it should retain from the previous tasks [11][35][33][1]. Such a hyperparameter can be optimized using any of the approaches described above, but it may also be used to perform trade-offs directly between tasks. For instance, when training on task 2, one could have a $\lambda_{2,1}$ that would be used to make trade-offs between how much the model should learn from task 2 and how much it should remember from task 1. When moving to task 3, one could have a

$\lambda_{3,1}$, to tune how much the model should learn from task 3 and how much it should remember from task 1 along with a $\lambda_{3,2}$ for how much the model should learn from task 3 versus how much it should remember from task 2. More generally, one could even set a different value for this hyperparameter for each pair of tasks, meaning that one could introduce $\lambda_{i,j}$ for mitigating catastrophic forgetting of task $j < i$ when learning task $i > 1$. Ultimately, this would result in a total of $\frac{(n-1)n}{2}$ hyperparameters. Similarly to the HPO with n hyperparameters, this way of performing HPO could fit the continual learning framework, see Tab. Table 2 bottom-right, but it could also be done in a cheated way, see Tab. Table 2 top-right.

Despite the choice and optimization of hyperparameters being critical, it seems that some confusion remain in the literature. The overwhelming majority of papers do not report their way of performing HPO or remain fuzzy about the management and values of certain hyperparameters [11][10][33]. On the other hand, among the papers that do report their HPO procedure, some fall into the cheated HPO category [14]. The end of this subsection will be dedicated to analyzing how HPO was likely conducted in a foundational paper in the literature that does not fully report the HPO procedure: the original Elastic Weight Consolidation (EWC) paper [11].

EWC is a regularisation method that attempts to tackle catastrophic forgetting by penalizing changes in important weights. The weights that are important for a task are the ones that updating would cause the model to move away from the solution of the task. Authors leverage the Fisher Information Matrix to define this importance and enforce the update of important weights to be performed in a direction that does not hurt the knowledge on the tasks already learnt. As introduced in the original paper, training on task 2 the loss used by EWC can be written as $\mathcal{L} = \mathcal{L}_2 + \lambda \mathcal{L}_{reg}$, where \mathcal{L}_2 accounts for maximizing the performance on task 2, λ is a positive scalar and \mathcal{L}_{reg} accounts for the regularization between task 1 and 2.

However, in this paper, authors do not explain how this regularization term scales with the number of tasks. It could be the case that the same value of λ is used for all the tasks, which would lead to write the loss used while training on task m as

$$\mathcal{L} = \mathcal{L}_m + \lambda L_{reg,m} \quad (1)$$

or it could be the case that a different value of λ is used for each task, which would lead to write the loss

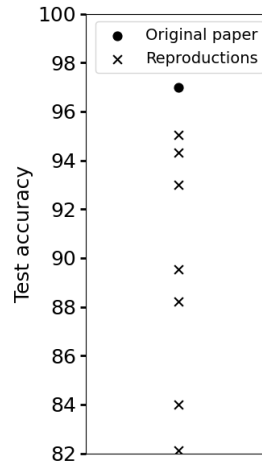


Figure 6: Summary of the results we identified in the literature with EWC on 10 tasks p-MNIST benchmarks [11][19][10][14][21][32][4][31]

used while training on task m as

$$\mathcal{L} = \mathcal{L}_m + \lambda_m L_{reg,m} \quad (2)$$

or the authors might want to make trade-offs between each pair of tasks, introducing a loss of the following form when training on task m

$$\mathcal{L} = \mathcal{L}_m + \sum_{i=1}^{m-1} \lambda_{m,i} \mathcal{L}_{reg,m,i} \quad (3)$$

Each of these 3 possible ways of proceeding would result in a total of $O(1)$, $O(n)$ or $O(n^2)$ hyperparameters, respectively. However, the authors do not provide any information about the way they proceeded, which is highly problematic. Additionally, authors do not report the value of the hyperparameter they introduced nor the process of optimizing it and they do not provide their code. This lack of clarity is a major issue as it makes it impossible to reproduce the results of the paper. Despite the fact that the community re-implemented the method, the method itself does not include the manipulation of hyperparameters made by authors. As a consequence, attempts to reproduce these methods are very disparate and yield contrasted results. See Fig. 6 for a summary of the results we identified in the literature.

Remarkably, all the papers reporting results above or equal to 90% accuracy on the 10 tasks p-MNIST benchmark are either performing a cheated HPO or not explaining their HPO procedure. In particular, the original paper mentions that the size of the hidden layers is an optimizable hyperparameter. However, the size

Number of HPs \ Number of HPOs	$O(1)$	$O(n)$	$O(n^2)$
1 cheated HPO	$\begin{array}{c} T_1 \dots T_n \\ \{\lambda\} \end{array}$ <i>HPO</i>	$\begin{array}{c} T_1 \dots T_n \\ \{\lambda_1, \dots, \lambda_n\} \end{array}$ <i>HPO</i>	$\begin{array}{c} T_1 \dots T_n \\ \{\lambda_{i,j} \mid \forall 2 \leq j < i \leq n\} \end{array}$ <i>HPO</i>
n greedy HPOs		$\begin{array}{c} T_1 \\ \{\lambda_1\} \end{array} \rightarrow \dots \rightarrow \begin{array}{c} T_n \\ \{\lambda_n\} \end{array}$ <i>HPO</i> ₁ <i>HPO</i> _{n}	$\begin{array}{c} T_2 \\ \{\lambda_{2,1}\} \end{array} \rightarrow \begin{array}{c} T_3 \\ \{\lambda_{3,1}, \lambda_{3,2}\} \end{array} \rightarrow \dots \rightarrow \begin{array}{c} T_n \\ \{\lambda_{n,1}, \dots, \lambda_{n,n-1}\} \end{array}$ <i>HPO</i> ₂ <i>HPO</i> ₃ <i>HPO</i> _{n}

Table 2: Various ways to perform HPOs in continual learning. For a given hyperparameter λ and tasks T_1, \dots, T_n , one can end-up with $O(1)$ (left column), $O(n)$ (middle column) or even $O(n^2)$ (right column) values of λ depending on the design of the HPO process. Additionally, one can perform a single HPO over the whole benchmark but it breaks continual learning assumptions (first line), or one can perform an HPO for each task (second line).

of the model remains fixed when training on the different tasks of the benchmark. This suggests that the HPO performed by the authors of the original paper covers the entire benchmark, that is, the HPO is likely to be cheated and include $O(n)$ hyperparameters. See Tab. Table 2 top-center.

5.2 Introduction of validation benchmarks

Actually, what we refer to as "cheated HPO" where a single HPO is performed through all the tasks at the same time, could be just fine if one was performing the HPO on a benchmark, and then showing the performances of the approach on another benchmark with the same hyperparameters. Such an approach would not even break the hypothesis according to which we do not know the number of tasks in advance. One would just have to perform the HPO on a benchmark with a huge number of tasks, and then show the performances of the approach on another benchmark, with less tasks, using only the hyperparameters computed for the first tasks of the HPO benchmark.

To better analyse methods in continual learning, we introduce a validation paradigm on benchmarks to measure overfitting on benchmarks. We perform HPO on a benchmark and then measure the performance on another benchmark on which we train the model using values of the HPs provided by the HPO performed on the HPO benchmark. However, one could argue that performing HPO on a benchmark like p-MNIST and validating the approach using optimized hyperparameters on a benchmark derived from CIFAR, for instance, would not be fair. And in order to avoid defining a brittle and handcrafted metric complexity or similarity between benchmarks, we will stick to p-MNIST benchmarks. Indeed, this validation paradigm we are introducing is one of the reasons why we chose to use p-MNIST benchmarks in the first place. Just as we did in Fig. 3, we generate 10 additional p-MNIST

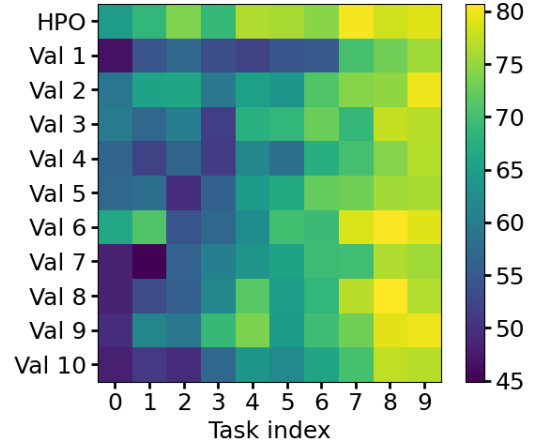


Figure 7: Test accuracy on each task once training with GroHess is complete on all the tasks, for the HPO benchmark and 10 validation benchmarks.

benchmarks, each with 10 tasks. As a consequence, we have benchmarks that share the same complexity and similarity from a statistical point of view. We do not perform any HPO on these validation benchmarks. Instead, we reuse the hyperparameter found on the previous benchmark, and simply retrain GroHess with them on the validation benchmarks. See Fig. 7 and Fig. 8 for the results of this validation paradigm.

For the HPO benchmark and each of the 10 validation benchmarks, Fig. 7 shows the test accuracy on each task once training with GroHess is complete on all the tasks. These results are run and averaged over seeds 88 to 92. At first glance, it seems that the performance on old tasks are better for the HPO benchmark, suggesting that it is harder for the model to mitigate catastrophic forgetting if the hyperparameters have not been specifically tuned on the benchmark. To confirm this, for each task, we compute the average test accuracy through the validation benchmarks (column wise average) and report the results in Fig. 8.

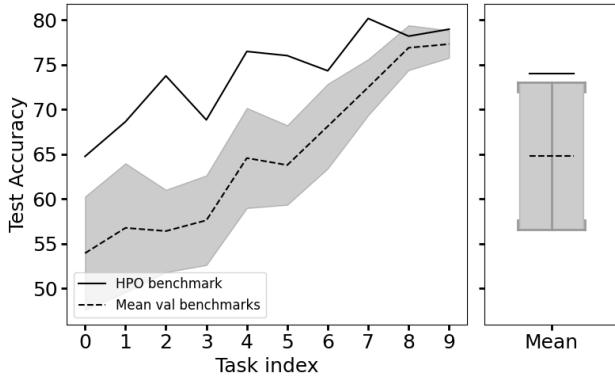


Figure 8: Results from Fig. 7 averaged over the validation benchmarks (left plot) and additionally averaged over all tasks (right plot).

In addition, this figure shows the standard deviation for performances on the validation benchmarks, along with the performances obtained on the HPO benchmark (left plot). The right plot shows the same results, averaged out over all tasks. We observe that the average test accuracy over all the tasks of the HPO benchmark are greater by an amount up to 9% than the average one on the validation benchmarks. This figure confirms that, despite the fact all the benchmarks have the same complexity, performing an HPO on a benchmark is necessary to get the best performances. In other words, GroHess lack of versatility and is overfitting on the benchmark through the HPO process.

These results show that the choice of hyperparameters is at the very core of the performances of our method. It means that catastrophic forgetting is mitigated not only through the network growth but also to a significant part through very specific arbitrations of how much the model should learn from new tasks as they come, that is, it seems that our method relies on trade-offs.

6 Critical comparison to literature

In the following section, we will develop remarks about the methods fitting with our assumptions: modularity, absence of memory and absence of constraints. See subsection 1.3 for recall of the assumptions. This section aims at investigating inherent limitations of GroHess and, more broadly, modular memoryless and unconstrained methods.

6.1 Trade-offs in continual learning

In the last section, we acknowledged that the hyperparameter optimization performed in the context of

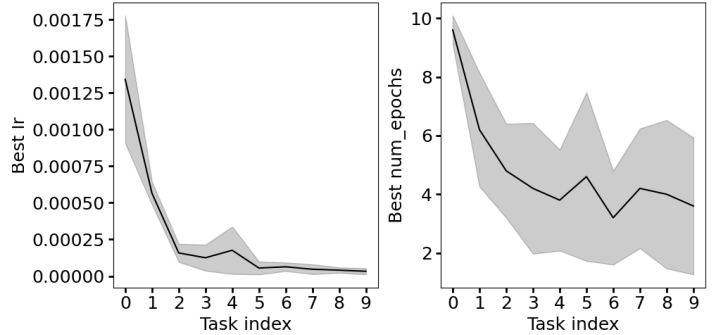


Figure 9: Best values of the hyperparameters (learning rate on the left and number of epochs on the right) found by the HPOs ran on each task of the HPO benchmark.

continual learning yields numerous hyperparameters. This can be an implicit way to perform trade-offs between how much the model should learn from the new task versus how much it should remember from the previous tasks. This lead has been confirmed through validation experiments, where we have seen that the very subtle differences between p-MNIST benchmarks are enough to impact the performances by an amount of 9%. As a consequence, we hypothesize that the HPO procedure is performing trade-offs between the tasks.

An interesting way to investigate this hypothesis is to have a look at the specific hyperparameters retained by the HPO. Fig. 9 shows the best values of the hyperparameters found by the HPOs ran on each task of the HPO benchmark. We observe that the learning rate is decreasing by over an order of magnitude as the model moves forward through the tasks. Similarly, we note that the best number of epochs used to train on a task was the highest allowed for the first tasks, and then quickly decreased for the latter tasks. As a consequence, the model is learning less and less of each tasks. Now we interpret this behavior as an attempt to keep track of previously acquired knowledge while learning on the new task. That is, our approach is mitigating catastrophic forgetting through trade-offs between the specificity of the tasks. As the differences between the specificity of the tasks rely on the permutations used to generate the tasks, our interpretation provides an explanation of why the model is overfitting on the HPO benchmark. Indeed, the trade-offs necessary to perform well on a given validation benchmark would require other hyperparameters than the ones found on the HPO benchmark.

Furthermore, in addition to making trade-offs between

learning and memory for each pair of tasks, methods like GroHess and those based on regularization, such as EWC, SI, LwF or AFEC [11][35][16][33], incorporate trade-offs within the optimization process. Indeed, the regularization term used in these methods is intended to perform trade-offs between the importance of the new task and the importance of the old tasks. They are controlled by a scalar, which is a hyperparameter is optimized through the HPO procedure. Similarly, through the computation of the Hessian and Gradient masks of importance, GroHess is trading off how much to update the weights that are the most crucial for the past tasks versus how much to update these weights for the benefit of the new task.

Even though all these methods are performing trade-offs, all the trade-offs are not the same. From one method to another, the hyperparameters are not the same, which means that the trade-offs that can be performed through HPO are not the same. This explains the disparities between the results of regularization methods gathered in Fig. 5b. Additionally, the trade-offs performed by the regularization methods are continuous, in the sense that they affect all the weights of the model more or less intensively. On the other hand, GroHess performs discrete trade-offs, only impacting a few weights, leaving the other ones completely unchanged. Finally, to explain results obtained with GroHess that happen to be even poorer than the ones yielded by our baselines, we note that our growing procedure is not optimal. Indeed, it is not embedded in an optimization procedure, as it is the case for the baselines and regularization methods who are optimized through back-propagation only. Thus, it is likely that our growing procedure is simply disturbing the learning process done through back-propagation.

6.2 Geometrical argument

In this subsection, we will discuss the geometrical motivation behind several regularization methods such as EWC [1][11]. Very often, the method is motivated by the representation of a 2-dimensional parameter space, where two ellipses represent regions of low loss when provided data from task A or task B. See Fig. 10. Once training on task A is done, one assumes that the model is in a state close to the argument of minimum of the loss for data associated to task A, θ_{A^*} . Then, the intuition is that the regularization term introduced in the method will prevent the parameters from converging directly toward the argument of the minimum of the loss for task 2, θ_{B^*} , when learning on task B. Instead, the parameters should converge to an overlapping re-

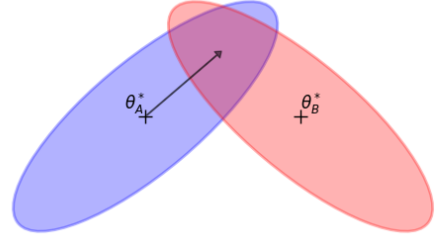


Figure 10: Representation of the parameters space as depicted in some regularization papers [11][1]. The blue and red ellipses represent a region with low loss on task A and task B, respectively.

gion where both loss on task A and loss on task B are small. However, this representation is misleading, does not reflect the complexity of the problem and, as a consequence, does not hold in practice.

A first objection is that with a regularization method, the direction of parameters modifications does not aim for a minimum of the model trained on the combined data from tasks A and B, but rather towards the local minimum of the model on task B’s data that is closest to the local minimum reached by the model trained solely on task A’s data.

Second, the Fisher Information Matrix introduced in EWC is leveraged to approximate a metric that indicates what is considered to be the best path to move toward low loss on task B, compromising with increase of loss on task A. However, loss on task A is not directly leveraged, as when training on task B, samples from task A are not accessible anymore. As a consequence, authors says that in order to stay close to the region of low loss on task A, it is reasonable to try to update the parameters in a direction of high curvature of the ellipse associated to task A. Indeed, the part of the ellipse that has the highest curvature is also the one that is the furthest from θ_{A^*} , which makes it a direction in which updates of the parameters can be carried without increasing the loss on task A too much. However, contrary to the representation in Fig. 10, this is not a guarantee that the model will converge to the center of the overlapping region. Indeed, the overlapping region might not be aligned with the axis passing through θ_{A^*} and the point of ellipse associated with task A with the highest curvature. See

Fig. 11. In these representation, the red arrow represent the update direction according to \mathcal{L}_B and the blue arrow represent the update direction enforced by \mathcal{L}_{reg} . The actual update direction is then decided by the value of λ and falls inside the cone generated by the two arrows, without covering the whole cone as the linear combination is done through a single scalar. Thus, what matters in the situations depicted in Fig. 10 is the angle between the two arrows along with the orientation of the blue arrow relatively to the red ellipse. The orientation of the red ellipse relatively to the blue arrow accounts for the success of the update: is the update going to land in a region where both loss on task A and on task B is low? On the other hand, the angle between the red and the blue arrows accounts for how much \mathcal{L}_B and \mathcal{L}_{reg} are acting jointly. In the case depicted in Fig. 11a, no matter what the value of λ is, the update will be performed toward the overlapping region. This means that the configuration depicted in EWC papers is a really convenient scenario. On the other hand, looking at the configuration in Fig. 11b, we see that the update will be performed toward the overlapping region only if the value of λ is low enough, because \mathcal{L}_{reg} is pulling the parameters away from the overlapping region. Ultimately, Fig. 11c depicts an extreme scenario, where \mathcal{L}_{reg} is actively acting against \mathcal{L}_B . In this configuration, any non zero value of λ would make the update worse both with respect to task A and B.

Furthermore, these illustrations represent a 2-dimensional parameters space, however, in the experiments conducted in the literature, the parameter space is \mathbb{R}^n with $n \sim 10^5$ or even orders of magnitudes bigger. Additionally, as we already mentioned earlier, the number of tasks in a continual learning benchmark is generally above or equal to 10, and the research community ultimately expects to be able to tackle as many as hundreds of tasks. As a consequence of these two remarks, the area (or volume) of the overlapping region tends doubly towards 0. On one hand, this representation accounts for two tasks, but in real life scenarios, there are many more. And as the number of tasks increases, the area of the overlapping region decreases, has 0 as a lower bound, so converges. But it has no reason to converge toward a non-zero value as the relative position of the low loss ellipses is *a priori* independent between the tasks, or at least, nobody has provided a better lower bound than 0, to the best of our knowledge. On the other hand, the previous remark implies that there exist a number of tasks for which the overlapping region of interest is small enough to fit within

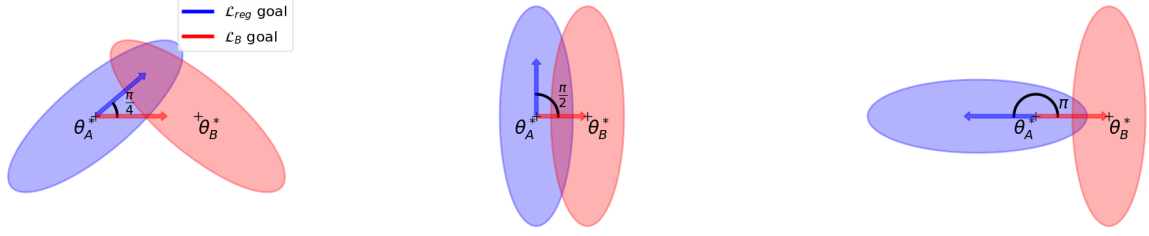
an Euclidean sphere whose radius is strictly smaller than 1, which volume, therefore, converges toward 0 as the dimension of the parameters space increases.

So, an increase in the number of tasks makes it less likely to find a region where the loss is low for all the tasks, while an increase in the number of parameters of the model somewhat pulls the low loss regions apart from each other. These two remarks do not constitute a strict argument against the regularization methods but rather bring to light the fact the motivation behind them is somewhat flawed. Additionally, they explain why literature report very poor performances of EWC on longer and harder benchmarks than p-MNIST [21][4]. Overall, our remarks mostly point out the absence of guarantees. The absence of guaranty is not dramatic though, as deep learning tends to be able to perform well in practice without strong theoretical results. But what is more problematic is that authors of modular, memoryless and unconstrained methods generally do not even have control over the object of our remarks, such as the relative position of the ellipses.

Finally, the worse inherent limitations of modular, memoryless and unconstrained methods is beautifully captured on the Fig. 10 :

*Improving on task B means deteriorating
on task A, and reciprocally.*

It does not mean that the more tasks there are to solve, the harder it gets to solve them all, which is true. Our statement is stronger, it means that tasks are inherently put in a competition setting with each other. This implies that performing continual learning with modular, memoryless and unconstrained methods inherently boils down to perform trade-offs, which confirms our prior analysis of trade-offs through HPOs. Ultimately, it implies that modular, memoryless and unconstrained methods are inherently limited when it comes to perform continual learning on numerous tasks. By contrast, we do not find this inherent limitation within Mixture of Experts models [8][5], where modules dedicated to each tasks are in parallel, and improving one particular module does not hurt another module as long as the gating system remains the same. And this gating system can be implemented using replay or rehearsal methods, or leveraging feature extractor such as CNN masks. But here we break one of our two assumptions: either we leverage some form of memory or we employ architectural constraints.



(a) Configuration depicted in EWC papers

(b) A configuration less favorable where both the angle and the orientation of the red ellipse are not convenient

(c) Extreme configuration where \mathcal{L}_{reg} is actively acting against \mathcal{L}_B

Figure 11: Several possible configurations in the parameters space. The blue and red ellipses represent a region with low loss on task A and task B , respectively. The red arrow represents the direction in which the loss on task B is pulling the model, while the blue arrow shows the direction in which the regularization term is pulling the model.

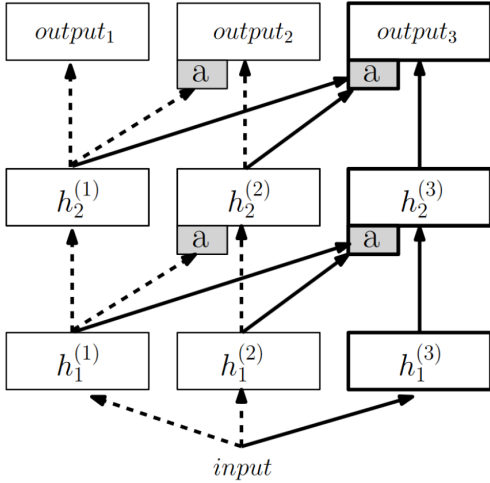


Figure 12: Extracted from [25]. Depiction of a three column progressive network. The first two columns on the left (dashed arrows) were trained on task 1 and 2 respectively. The grey box labelled a represent the adapter layers (see text). A third column is added for the final task having access to all previously learned features

6.3 Combinatorial argument

In this subsection, we will provide a combinatorial point of view on the inherent limitations of modular, memoryless and unconstrained methods.

Modular approaches are based on the idea that the model can be decomposed into several modules, each of them being dedicated to a task. Working with such methods in the context of continual learning raises three questions: how to distribute a given sample to the right module? How to specialize the modules?

How to merge the outputs of the modules? The first question could be answered by a gating system, which, however, does not respect our memoryless and unconstrained assumptions. The second question is well studied and several satisfying answers are provided in the literature. Modules can be specialized through regularization or may even be trained separately from one another. The third question is generally answered by a form of weighted linear combination of the outputs of the modules, where the weights of the linear combination are optimized through the HPO procedure. And in the answer to this last question lies the inherent limitation of modular, memoryless and unconstrained that we already put forward in the previous section: improving on a task (i.e. putting a higher weight when merging its associated module) means deteriorating on another one, and reciprocally.

An example of method that does not circumvent this limitation is called Progressive Neural Networks (PNN) [25]. This method falls in the category of architectural approaches to continual learning. In PNN, the model is being grown every time a new task is tackled. The grown part consists of a whole deep neural network. The distribution of task is done through adapter layers which are trained to forward a sample to the next layer only if belongs to the right task. See Fig. 12.

Each column is trained on the data of the task it is intended to solve. It also does take into account the learning of the previous tasks. However, a given column is unable to deal with samples of the latter tasks. As a consequence, a PNN may be able to solve each task, through one of the columns, once the training on

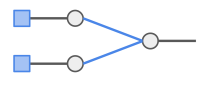
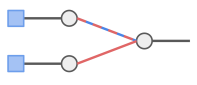
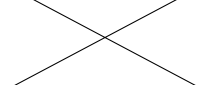
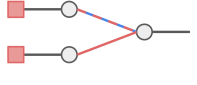
Trained \ Tested	Task 1	Task 2
Task 1		
Task 2		

Table 3: After training on task 1, there is no pollution (top left). After training on task 2, there is $P^{1 \rightarrow 2}$, pollution from task-2-specific weights when forwarding task 1 samples (top right) and $P^{2 \rightarrow 1}$, pollution from task-1-specific weights when forwarding task 2 samples (bottom right).

all the tasks is completed. However, it remains unable to choose properly which column for the inference of a sample from a given tasks.

The problem faced by PNN is very general, we refer to it by the term of *pollution*, a concept that we will attempt to formalize below.

Let us assume we are given I tasks indexed in increasing order by the set $\llbracket 1, I \rrbracket$. We define a neural network as the set of its parameters

$$\phi = \{ \{ w_{l,m,n} \in \mathbb{R} | m, n \in \llbracket 1, N_{l-1} \rrbracket \times \llbracket 1, N_l \rrbracket \} | l \in \llbracket 1, L+1 \rrbracket \} \quad (4)$$

where L is the number of hidden layers and N_l is the number of neurons on layer l .

Let us define $\mathbb{W}_{l,n}^i$ to be the set of protected weights for task i with respect to neuron n of layer l , for any $i \in \llbracket 1, I \rrbracket$, $l \in \llbracket 1, L+1 \rrbracket$ and $n \in \llbracket 1, N_l \rrbracket$:

$$\mathbb{W}_{l,n}^i = \{ w_{l,m,n}, m \in \llbracket 1, N_{l-1} \rrbracket | w_{l,m,n} \text{ protected} \} \quad (5)$$

which allows to define the pollution of task i by protected weights for task j on neuron n of layer l as

$$p_{l,n}^{i \rightarrow j} = \mathbb{E}_{x \sim D_i} \left[\frac{\sum_{w \in \mathbb{W}_{l,n}^j} w(x)}{\sum_{w \in \mathbb{W}_{l,n}^i} w(x)} \right] \quad (6)$$

Finally, we can define the pollution of task i by protected weights for task j on the whole network as

$$P^{i \rightarrow j} = \frac{1}{L+1} \sum_{l=1}^{L+1} \frac{1}{N_l} \sum_{n=1}^{N_l} p_{l,n}^{i \rightarrow j} \quad (7)$$

The notion of pollution $P^{i \rightarrow j}$ encapsulates how much neurons specialized for a task j interfere with neurons

specialized for a task i when provided a sample from task i , after training on the first $\max(i, j)$ tasks at least. See Tab. Table 3 for a simple illustration. Once the model is trained on task 1, we can test it on data from task 1 (top left cell) but we can't expect it to perform well on data from the next tasks (bottom left cell), so no pollution is occurring. Then, suppose the upper blue line to represent a module of a neuron that we want to dedicate to task 1. This blue line is the module specialized to task 1, we won't update it anymore. Once the model is trained on task 2, we expect it to perform well on task 1 and 2, so the pollutions that can occur are $P^{1 \rightarrow 2}$ (top right cell) and $P^{2 \rightarrow 1}$ (bottom right cell). $P^{2 \rightarrow 1}$ is already problematic, but is generally handled through trade-offs as we mentioned earlier. When the model is being trained on task 2, it learns how to deal with the outputs of the module specialized for task 1 when given samples from task 2. $P^{2 \rightarrow 1}$ is the pollution that can be tackled through trade-offs. However, when training on task 2 is done, the lower red module in the top right cell has been overwritten by optimization on samples of task 2 only. As consequence when provided a samples from task 1 again, during test time, the upper dedicated blue module might output reasonably well, however, the lower module, now specialized on task 2, does not know how to deal with samples from task 1. That is why $P^{1 \rightarrow 2}$ cannot be handled by modular, memoryless and unconstrained methods. More broadly, the real challenge is to cope with $P^{i \rightarrow j}$ for $i < j$.

Overall, ideally, one would like the merging of the modules to be binary, and only consider the output of the module dedicated to the task of the sample at hand, during inference. This means being able to identify the task of the sample at hand. So, in this sense, answering the third question (merging) boils down to answer the first question (distributing), which cannot be answered under our hypotheses. In the absence of a way to circumvent this issue, approaches have no choice but to perform trade-offs between the tasks, which causes their limited performances.

7 Conclusion

Overall, in this thesis, in addition to explore and develop a method to leverage artificial neurogenesis in order to tackle continual learning, we did an extensive and critical review of a branch of literature of continual learning. This critic included both a geometrical and a combinatorial argument which both lead to the conclusion that many regularization-based and

architectural-based approaches are inherently limited to perform trade-offs, which explains the comparative success of other approaches focused on memory and emerging features. As a result, we recommend moving away from modular methods that neither utilize any form of memory from the past tasks nor leverage architectural constraints to enable the emergence and manipulation of task’s features.

Additionally, we acknowledged that the manipulation of hyperparameters in the context of continual learning is subtle, and we encourage more care in their management. Finally, along with the study of hyperparameters optimizations, the introduction of a validation paradigm in continual learning led us to identify a form of overfitting on benchmarks, and ultimately led us to the conclusion that the performances of several continual learning approaches, including ours, are highly dependent on the choice of hyperparameters, which confirms that trade-offs are a core component of the success of these approaches.

8 Acknowledgment

I would like to express my deepest gratitude to Melika Payvand for offering the great opportunity to do my Master thesis in her team. Her guidance and encouragement throughout this thesis have been crucial in its completion. I am also thankful Karthik Raghunathan, whose idea GroHess was. Numerous technical discussions with him allowed me develop this method in its entirety. I also want to warmly thank my entire team, whose feedback has truly shaped my thesis into what it is today and who made the time spent with them an enjoyable period that I will remember. I am also grateful to the whole Institute of Neuroinformatic (INI) for welcoming me. Beyond my thesis, the various fascinating discussions I had with members of INI broadened my perspective on Research, as a general concept, and helped me to clarify my own specific interests. A special thank you goes to Guillaume Charpiat for accepting to be my referent professor, and providing meaningful support every time I needed it. Lastly, I would like to express my deepest appreciation to my parents, for their unwavering support throughout this thesis and throughout my entire studies. Without them, this journey would not have been possible.

References

- [1] Abhishek Aich. *Elastic Weight Consolidation (EWC): Nuts and Bolts*. 2021. arXiv: 2105.04093.
- [2] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019.
- [3] Rahaf Aljundi et al. *Memory Aware Synapses: Learning what (not) to forget*. 2018. arXiv: 1711.09601 [cs.CV]. URL: <https://arxiv.org/abs/1711.09601>.
- [4] Sayna Ebrahimi et al. *Adversarial Continual Learning*. 2020. arXiv: 2003.09553 [cs.LG]. URL: <https://arxiv.org/abs/2003.09553>.
- [5] David Eigen, Marc’Aurelio Ranzato, and Ilya Sutskever. *Learning Factored Representations in a Deep Mixture of Experts*. 2014. arXiv: 1312.4314 [cs.LG].
- [6] Sebastian Farquhar and Yarin Gal. *Towards Robust Evaluations of Continual Learning*. 2019. arXiv: 1805.09733 [stat.ML]. URL: <https://arxiv.org/abs/1805.09733>.
- [7] Ian J. Goodfellow et al. *An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks*. 2015. arXiv: 1312.6211 [stat.ML]. URL: <https://arxiv.org/abs/1312.6211>.
- [8] Isobel Claire Gormley and Sylvia Frühwirth-Schnatter. *Mixtures of Experts Models*. 2018. arXiv: 1806.08200 [stat.ME]. URL: <https://arxiv.org/abs/1806.08200>.
- [9] Yen-Chang Hsu et al. *Re-evaluating Continual Learning Scenarios: A Categorization and Case for Strong Baselines*. 2019. arXiv: 1810.12488 [cs.LG]. URL: <https://arxiv.org/abs/1810.12488>.
- [10] Ronald Kemker et al. *Measuring Catastrophic Forgetting in Neural Networks*. 2017. arXiv: 1708.02072.
- [11] James Kirkpatrick et al. “Overcoming catastrophic forgetting in neural networks”. In: *Proceedings of the National Academy of Sciences* 114.13 (Mar. 2017), pp. 3521–3526. ISSN: 1091-6490. DOI: 10.1073/pnas.1611835114. URL: <http://dx.doi.org/10.1073/pnas.1611835114>.

- [12] Dhireesha Kudithipudi et al. “Biological underpinnings for lifelong learning machines”. In: *Nature Machine Intelligence* 4.3 (Mar. 2022), pp. 196–210. ISSN: 2522-5839. DOI: 10.1038/s42256-022-00452-0. URL: <https://doi.org/10.1038/s42256-022-00452-0>.
- [13] Dhireesha Kudithipudi et al. “Design principles for lifelong learning AI accelerators”. In: *Nature Electronics* 6.11 (Nov. 2023), pp. 807–822. ISSN: 2520-1131. DOI: 10.1038/s41928-023-01054-3. URL: <https://doi.org/10.1038/s41928-023-01054-3>.
- [14] Alexey Kutalev and Alisa Lapina. *Stabilizing Elastic Weight Consolidation method in practical ML tasks and using weight importances for neural network pruning*. 2021. arXiv: 2109.10021 [cs.LG]. URL: <https://arxiv.org/abs/2109.10021>.
- [15] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.
- [16] Zhizhong Li and Derek Hoiem. *Learning without Forgetting*. 2017. arXiv: 1606.09282 [cs.CV]. URL: <https://arxiv.org/abs/1606.09282>.
- [17] David Lopez-Paz and Marc’Aurelio Ranzato. *Gradient Episodic Memory for Continual Learning*. 2022. arXiv: 1706.08840 [cs.LG]. URL: <https://arxiv.org/abs/1706.08840>.
- [18] Davide Maltoni and Vincenzo Lomonaco. “Continuous Learning in Single-Incremental-Task Scenarios”. In: *CoRR* abs/1806.08568 (2018). arXiv: 1806.08568. URL: <http://arxiv.org/abs/1806.08568>.
- [19] Cuong V. Nguyen et al. “Variational Continual Learning”. In: *International Conference on Learning Representations*. 2018. URL: <https://openreview.net/forum?id=BkQqq0gRb>.
- [20] German Ignacio Parisi et al. “Continual Lifelong Learning with Neural Networks: A Review”. In: *CoRR* abs/1802.07569 (2018). arXiv: 1802.07569. URL: <http://arxiv.org/abs/1802.07569>.
- [21] Krishnan Raghavan and Prasanna Balaprakash. “Formalizing the Generalization-Forgetting Trade-off in Continual Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 17284–17297. URL: [https://proceedings.neurips.cc/paper_files/paper/2021/file/](https://proceedings.neurips.cc/paper_files/paper/2021/file/901797aebf0b23ecbab534d61ad33bb1-Paper.pdf)
- 901797aebf0b23ecbab534d61ad33bb1 – Paper .pdf.
- [22] Sylvestre-Alvise Rebuffi et al. *iCaRL: Incremental Classifier and Representation Learning*. 2017. arXiv: 1611.07725 [cs.CV]. URL: <https://arxiv.org/abs/1611.07725>.
- [23] David Rolnick et al. “Experience Replay for Continual Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/fa7cdfad1a5aaf8370ebeda47a-Paper.pdf.
- [24] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: <https://doi.org/10.1038/323533a0>.
- [25] Andrei A. Rusu et al. *Progressive Neural Networks*. 2022. arXiv: 1606.04671 [cs.LG]. URL: <https://arxiv.org/abs/1606.04671>.
- [26] Joan Serra et al. *Overcoming catastrophic forgetting with hard attention to the task*. 2018. arXiv: 1801.01423 [cs.LG]. URL: <https://arxiv.org/abs/1801.01423>.
- [27] Hanul Shin et al. *Continual Learning with Deep Generative Replay*. 2017. arXiv: 1705.08690 [cs.AI]. URL: <https://arxiv.org/abs/1705.08690>.
- [28] Hanul Shin et al. “Continual Learning with Deep Generative Replay”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/0efbe98067c6c73dba1250d2b-Paper.pdf.
- [29] Guido M. van de Ven, Hava T. Siegelmann, and Andreas S. Tolias. “Brain-inspired replay for continual learning with artificial neural networks”. In: *Nature Communications* 11.1 (Aug. 2020), p. 4069. ISSN: 2041-1723. DOI: 10.1038/s41467-020-17866-2. URL: <https://doi.org/10.1038/s41467-020-17866-2>.
- [30] Guido M. van de Ven and Andreas S. Tolias. *Generative replay with feedback connections as a general strategy for continual learning*. 2019. arXiv: 1809.10635 [cs.LG]. URL: <https://arxiv.org/abs/1809.10635>.

- [31] Guido M. van de Ven and Andreas S. Tolias. *Three scenarios for continual learning*. 2019. arXiv: 1904.07734 [cs.LG]. URL: <https://arxiv.org/abs/1904.07734>.
- [32] Usman Wajid et al. “A three-way decision approach for dynamically expandable networks”. In: *International Journal of Approximate Reasoning* 166 (2024), p. 109105. ISSN: 0888-613X. DOI: <https://doi.org/10.1016/j.ijar.2023.109105>. URL: <https://www.sciencedirect.com/science/article/pii/S0888613X23002360>.
- [33] Liyuan Wang et al. *AFEC: Active Forgetting of Negative Transfer in Continual Learning*. 2021. arXiv: 2110.12187.
- [34] Jaehong Yoon et al. *Lifelong Learning with Dynamically Expandable Networks*. 2018. arXiv: 1708.01547 [cs.LG]. URL: <https://arxiv.org/abs/1708.01547>.
- [35] Friedemann Zenke, Ben Poole, and Surya Ganguli. *Continual Learning Through Synaptic Intelligence*. 2017. arXiv: 1703.04200 [cs.LG]. URL: <https://arxiv.org/abs/1703.04200>.